目錄

简介

Python进阶

《Python进阶》是《Intermediate Python》的中文译本, 谨以此献给进击的 Python 和 Python 程序员们!

快速阅读传送门

- Github快速阅读任一章节：进入目录
- Gitbook完整顺序地阅读：进入Gitbook
- 本地或kindle上阅读：下载pdf/epub/mobi
- 国内推荐镜像（实时同步）：极客学院收录
- 其他镜像（不定期同步）：Pythontab收录
- 纯代码阅读和演示：进入code目录

前言

Python，作为一个"老练"、"小清新"的开发语言，已受到广大才男俊女的喜爱。我们也从最基础的Python粉，经过时间的摧残慢慢的变成了Python老鬼。

IntermediatePython这本书具有如下几个优点：

1. 简单
2. 易读
3. 易译

这些都不是重点，重点是：它是一本开脑洞的书。无论你是Python初学者，还是Python高手，它显现给你的永远是Python里最美好的事物。

> 世上语言千万种 美好事物藏其中

译者在翻译过程中，慢慢发现，本书作者的行文方式有着科普作家的风范，--那就是能将晦涩难懂的技术用比较清晰简洁的方式进行呈现，深入浅出的风格在每个章节的讨论中都得到了体现：

- 每个章节都非常精简，5分钟就能看完，用最简洁的例子精辟地展现了原理
- 每个章节都会通过疑问，来引导读者主动思考答案
- 每个章节都引导读者做延伸阅读，让有兴趣的读者能进一步举一反三
- 每个章节都是独立的，你可以挑选任意的章节开始阅读，而不受影响

总之，这本书非常方便随时选取一个章节进行阅读，而且每次阅读一个章节，你都可能会有一些新的发现。

原书作者

感谢英文原著作者 @yasoob《Intermediate Python》，有了他才有了这里的一切

译者

老高 @spawnris
刘宇 @liuyu
明源 @muxueqz
大牙matt @suqi
蒋委员长 @jiedo

欢迎建议指正或直接贡献代码

https://github.com/eastlakeside/interpy-zh/issues

微信交流群

微信打赏支持：

微信扫一扫 支付

序

序

这是一本Intermediate Python 的中文译本, 谨以此献给进击的 Python 和 Python 程序员们!

这是一次团队建设、一次尝鲜、一次对自我的提升。相信每个有为青年，心里想藏着一个小宇宙：我想要做一件有意思的事。$$什么是有意思的事？$$ 别闹

Python，作为一个"老练"、"小清新"的开发语言，已受到广大才男俊女的喜爱。我们也从最基础的Python粉，经过时间的摧残慢慢的变成了Python老鬼。因此一开始 @大牙 提出要翻译点什么的时候，我还是挺兴奋的，团队一起协作，不单可以磨练自己，也能加强团队之间的协作。为此在经过短暂的讨论后，翻译的内容就定为：《Intermediate Python》。

IntermediatePython这本书具有如下几个优点：

1. 简单
2. 易读
3. 易译

这些都不是重点，重点是：它是一本开脑洞的书。无论你是Python初学者，还是Python高手，它显现给你的永远是Python里最美好的事物。

> 世上语言千万种 美好事物藏其中

翻译的过程很顺利，语言很易懂，因此各位读者欢迎捐赠，或加入微信群讨论。

译后感

译者后记

译者大牙感言

在翻译过程中，慢慢发现，本书作者的行文方式有着科普作家的风范，--那就是能将晦涩难懂的技术用比较清晰简洁的方式进行呈现，深入浅出的风格在每个章节的讨论中都得到了体现：

- 每个章节都非常精简，5分钟就能看完，用最简洁的例子精辟地展现了原理
- 每个章节都会通过疑问，来引导读者主动思考答案
- 每个章节都引导读者做延伸阅读，让有兴趣的读者能进一步举一反三
- 每个章节都是独立的，你可以挑选任意的章节开始阅读，而不受影响

总之，这本书非常方便随时选取一个章节进行阅读，而且每次阅读一个章节，你都可能会有一些新的发现。

本译作已开源，欢迎**PullRequest**

- gitbook: https://eastlakeside.gitbooks.io/interpy-zh/
- github: https://github.com/eastlakeside/interpy-zh
- gitbook 与 github 已绑定，会互相同步

原作者前言

关于原作者

我是 Muhammad Yasoob Ullah Khalid.

我已经广泛使用 Python 编程3年多了. 同时参与了很多开源项目. 并定期在我的博客里写一些关于Python有趣的话题.

2014年我在柏林举办的欧洲最大的Python会议**EuroPython**上做过精彩的演讲.

> 译者注：分享的主题为：《Session: Web Scraping in Python 101》 地址：https://ep2014.europython.eu/en/schedule/sessions/20/

如果你能给我有意思的工作机会, 请联系我哦.

> 译者注：嗯，硬广，你来中国么，HOHO

作者前言

Hello 大家好! 我非常自豪地宣布我自己创作的书完成啦.
经过很多辛苦工作和决心, 终于将不可能变成了可能, "Intermediate Python"终于杀青.
ps: 它还将持续更新 :)

Python 是一门奇妙的语言, 还有一个强大而友爱的程序员社区.
然而, 在你理解消化掉 Python 的基础后, 关于下一步学习什么的资料比较缺乏. 而我正是要通过本书来解决这一问题. 我会给你一些可以进一步探索的有趣的话题的信息.

本书讨论的这些话题将会打开你的脑洞, 将你引导至 Python 语言的一些美好的地方. 我最开始学习 Python 时, 渴望见到Python最优雅的地方, 而本书正是这些渴望的结果.

无论你是个初学者, 中级或者甚至高级程序员, 你都会在这本书里有所收获.

请注意本书不是一个指导手册, 也不会教你 Python. 因为书中的话题并没有进行基础解释, 而只提供了展开讨论前所需的最少信息.

好啦，你肯定也和我一样兴奋, 那让我们开始吧!

开源公告

注意: 这本书是开源的, 也是一个持续进展中的工作. 如果你发现typo, 或者想添加更多内容进来, 或者可以改进的任意地方(我知道你会发现很多), 那么请慷慨地提交一个 pull request, 我会无比高兴地合并进来. :)

另外, 我决定将这本书免费发布! 我相信它会帮助到那些需要帮助的人. 祝你们好运!

这里是免费阅读链接:

- Html

- PDF
- GitHub

广告

注意: 你也可以现在为我捐助, 如果你想买Gumroad 提供的高大上版本.

你也可以加入我的邮件列表, 这样你可以保持同步获取到重大更新或者我未来其他项目!

最后而且也很重要的是, 如果你读了这本书, 并且发现它很有帮助, 那么一个私人邮件和一个 tweet 分享, 对我来说会很有意义.

args 和 *kwargs

*args 和 **kwargs

我观察到，大部分新的Python程序员都需要花上大量时间理解清楚 *args 和**kwargs这两个魔法变量。那么它们到底是什么?

首先让我告诉你, 其实并不是必须写成*args 和**kwargs。 只有变量前面的 *(星号)才是必须的. 你也可以写成*var 和**vars. 而写成*args 和**kwargs只是一个通俗的命名约定。那就让我们先看一下*args吧。

*args 的用法

*args 的用法

*args 和 **kwargs 主要用于函数定义。 你可以将不定数量的参数传递给一个函数。

这里的不定的意思是：预先并不知道, 函数使用者会传递多少个参数给你, 所以在这个场景下使用这两个关键字。 *args 是用来发送一个非键值对的可变数量的参数列表给一个函数.

这里有个例子帮你理解这个概念:

```python
def test_var_args(f_arg, *argv):
    print("first normal arg:", f_arg)
    for arg in argv:
        print("another arg through *argv:", arg)

test_var_args('yasoob', 'python', 'eggs', 'test')
```

这会产生如下输出:

```
first normal arg: yasoob
another arg through *argv: python
another arg through *argv: eggs
another arg through *argv: test
```

我希望这解决了你所有的困惑. 那接下来让我们谈谈 **kwargs

**kwargs 的用法

**kwargs 的用法

kwargs 允许你将不定长度的键值对, 作为参数传递给一个函数。 如果你想要在一个函数里处理带名字的参数, 你应该使用kwargs。

这里有个让你上手的例子:

```
def greet_me(**kwargs):
    for key, value in kwargs.items():
        print("{0} == {1}".format(key, value))


>>> greet_me(name="yasoob")
name == yasoob
```

现在你可以看出我们怎样在一个函数里, 处理了一个键值对参数了。

这就是**kwargs的基础, 而且你可以看出它有多么管用。 接下来让我们谈谈，你怎样使用*args 和 **kwargs来调用一个参数为列表或者字典的函数。

使用 args 和 *kwargs 来调用函数

使用 *args 和 **kwargs 来调用函数

那现在我们将看到怎样使用*args和**kwargs 来调用一个函数。 假设，你有这样一个小函数：

```
def test_args_kwargs(arg1, arg2, arg3):
    print("arg1:", arg1)
    print("arg2:", arg2)
    print("arg3:", arg3)
```

你可以使用*args或**kwargs来给这个小函数传递参数。 下面是怎样做：

```
# 首先使用 *args
>>> args = ("two", 3, 5)
>>> test_args_kwargs(*args)
arg1: two
arg2: 3
arg3: 5

# 现在使用 **kwargs:
>>> kwargs = {"arg3": 3, "arg2": "two", "arg1": 5}
>>> test_args_kwargs(**kwargs)
arg1: 5
arg2: two
arg3: 3
```

标准参数与*args、**kwargs在使用时的顺序

那么如果你想在函数里同时使用所有这三种参数， 顺序是这样的：

```
some_func(fargs, *args, **kwargs)
```

啥时候使用它们

什么时候使用它们？

这还真的要看你的需求而定。

最常见的用例是在写函数装饰器的时候（会在另一章里讨论）。

此外它也可以用来做猴子补丁(monkey patching)。猴子补丁的意思是在程序运行时(runtime)修改某些代码。 打个比方，你有一个类，里面有个叫`get_info`的函数会调用一个API并返回相应的数据。如果我们想测试它，可以把API调用替换成一些测试数据。例如：

```
import someclass

def get_info(self, *args):
    return "Test data"

someclass.get_info = get_info
```

我敢肯定你也可以想象到一些其他的用例。

调试 Debugging

调试（Debugging）

利用好调试，能大大提高你捕捉代码Bug的。大部分新人忽略了Python debugger(pdb)的重要性。 在这个章节我只会告诉你一些重要的命令，你可以从官方文档中学习到更多。

> 译者注，参考：https://docs.python.org/2/library/pdb.html Or https://docs.python.org/3/library/pdb.html

从命令行运行

你可以在命令行使用Python debugger运行一个脚本， 举个例子：

```
$ python -m pdb my_script.py
```

这会触发debugger在脚本第一行指令处停止执行。这在脚本很短时会很有帮助。你可以通过(Pdb)模式接着查看变量信息，并且逐行调试。

从脚本内部运行

同时，你也可以在脚本内部设置断点，这样就可以在某些特定点查看变量信息和各种执行时信息了。这里将使用`pdb.set_trace()`方法来实现。举个例子：

```
import pdb

def make_bread():
    pdb.set_trace()
    return "I don't have time"

print(make_bread())
```

试下保存上面的脚本后运行之。你会在运行时马上进入debugger模式。现在是时候了解下debugger模式下的一些命令了。

命令列表：

- c: 继续执行
- w: 显示当前正在执行的代码行的上下文信息
- a: 打印当前函数的参数列表
- s: 执行当前代码行，并停在第一个能停的地方（相当于单步进入）
- n: 继续执行到当前函数的下一行，或者当前行直接返回（单步跳过）

单步跳过（next）和单步进入（step）的区别在于， 单步进入会进入当前行调用的函数内部并停在里面， 而单步跳过会（几乎）全速执行完当前调用的函数，并停在当前函数的下一行。

pdb真的是一个很方便的功能，上面仅列举少量用法，更多的命令强烈推荐你去看官方文档。

生成器 Generators

生成器（Generators）

首先我们要理解迭代器(iterators)。根据维基百科，迭代器是一个让程序员可以遍历一个容器（特别是列表）的对象。然而，一个迭代器在遍历并读取一个容器的数据元素时，并不会执行一个迭代。你可能有点晕了，那我们来个慢动作。换句话说这里有三个部分：

- 可迭代对象(Iterable)
- 迭代器(Iterator)
- 迭代(Iteration)

上面这些部分互相联系。我们会先各个击破来讨论他们，然后再讨论生成器(generators).

可迭代对象(Iterable)

可迭代对象(Iterable)

Python中任意的对象，只要它定义了可以返回一个迭代器的`__iter__`方法，或者定义了可以支持下标索引的`__getitem__`方法(这些双下划线方法会在其他章节中全面解释)，那么它就是一个可迭代对象。简单说，可迭代对象就是能提供迭代器的任意对象。那迭代器又是什么呢？

迭代器(Iterator)

迭代器(Iterator)

任意对象，只要定义了`next`(Python2) 或者`__next__`方法，它就是一个迭代器。就这么简单。现在我们来理解迭代(iteration)

迭代(Iteration)

迭代(Iteration)

用简单的话讲，它就是从某个地方（比如一个列表）取出一个元素的过程。当我们使用一个循环来遍历某个东西时，这个过程本身就叫迭代。现在既然我们有了这些术语的基本理解，那我们开始理解生成器吧。

生成器(Generators)

生成器(Generators)

生成器也是一种迭代器，但是你只能对其迭代一次。这是因为它们并没有把所有的值存在内存中，而是在运行时生成值。你通过遍历来使用它们，要么用一个“for”循环，要么将它们传递给任意可以进行迭代的函数和结构。大多数时候生成器是以函数来实现的。然而，它们并不返回一个值，而是`yield`(暂且译作“生出”)一个值。这里有个生成器函数的简单例子：

```
def generator_function():
    for i in range(10):
        yield i

for item in generator_function():
    print(item)

# Output: 0
# 1
# 2
# 3
# 4
# 5
# 6
# 7
# 8
# 9
```

这个案例并不是非常实用。生成器最佳应用场景是：你不想同一时间将所有计算出来的大量结果集分配到内存当中，特别是结果集里还包含循环。

> 译者注：这样做会消耗大量资源

许多Python 2里的标准库函数都会返回列表，而Python 3都修改成了返回生成器，因为生成器占用更少的资源。

下面是一个计算斐波那契数列的生成器：

```
# generator version
def fibon(n):
    a = b = 1
    for i in range(n):
        yield a
        a, b = b, a + b
```

函数使用方法如下：

```
for x in fibon(1000000):
    print(x)
```

用这种方式，我们可以不用担心它会使用大量资源。然而，之前如果我们这样来实现的话：

```
def fibon(n):
    a = b = 1
    result = []
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result
```

这也许会在计算很大的输入参数时，用尽所有的资源。我们已经讨论过生成器使用一次迭代，但我们并没有测试过。在测试前你需要再知道一个Python内置函数：`next()`。它允许我们获取一个序列的下一个元素。那我们来验证下我们的理解：

```
def generator_function():
    for i in range(3):
        yield i

gen = generator_function()
print(next(gen))
# Output: 0
print(next(gen))
# Output: 1
print(next(gen))
# Output: 2
print(next(gen))
# Output: Traceback (most recent call last):
#            File "<stdin>", line 1, in <module>
#         StopIteration
```

我们可以看到，在`yield`掉所有的值后，`next()`触发了一个`StopIteration`的异常。基本上这个异常告诉我们，所有的值都已经被`yield`完了。你也许会奇怪，为什么我们在使用`for`循环时没有这个异常呢？啊哈，答案很简单。`for`循环会自动捕捉到这个异常并停止调用`next()`。你知不知道Python中一些内置数据类型也支持迭代哦？我们这就去看看：

```
my_string = "Yasoob"
next(my_string)
# Output: Traceback (most recent call last):
#      File "<stdin>", line 1, in <module>
#    TypeError: str object is not an iterator
```

好吧，这不是我们预期的。这个异常说那个`str`对象不是一个迭代器。对，就是这样！它是一个可迭代对象，而不是一个迭代器。这意味着它支持迭代，但我们不能直接对其进行迭代操作。那我们怎样才能对它实施迭代呢？是时候学习下另一个内置函数，`iter`。它将根据一个可迭代对象返回一个迭代器对象。这里是我们如何使用它：

```
my_string = "Yasoob"
my_iter = iter(my_string)
next(my_iter)
# Output: 'Y'
```

现在好多啦。我肯定你已经爱上了学习生成器。一定要记住，想要完全掌握这个概念，你只有使用它。确保你按照这个模式，并在生成器对你有意义的任何时候都使用它。你绝对不会失望的！

Map，Filter 和 Reduce

Map，Filter 和 Reduce

Map，Filter 和 Reduce 三个函数能为函数式编程提供便利。我们会通过实例一个一个讨论并理解它们。

Map

Map

Map会将一个函数映射到一个输入列表的所有元素上。这是它的规范：

规范

```
map(function_to_apply, list_of_inputs)
```

大多数时候，我们要把列表中所有元素一个个地传递给一个函数，并收集输出。比方说：

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
```

Map可以让我们用一种简单而漂亮得多的方式来实现。就是这样：

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
```

大多数时候，我们使用匿名函数(lambdas)来配合map, 所以我在上面也是这么做的。 不仅用于一列表的输入， 我们甚至可以用于一列表的函数！

```
def multiply(x):
        return (x*x)
def add(x):
        return (x+x)

funcs = [multiply, add]
for i in range(5):
    value = map(lambda x: x(i), funcs)
    print(list(value))
    # 译者注：上面print时，加了list转换，是为了python2/3的兼容性
    #        在python2中map直接返回列表，但在python3中返回迭代器
    #        因此为了兼容python3, 需要list转换一下

# Output:
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

Filter

Filter

顾名思义，filter过滤列表中的元素，并且返回一个由所有符合要求的元素所构成的列表，符合要求即函数映射到该元素时返回值为True. 这里是一个简短的例子：

```
number_list = range(-5, 5)
less_than_zero = filter(lambda x: x < 0, number_list)
print(list(less_than_zero))
# 译者注：上面print时，加了list转换，是为了python2/3的兼容性
#        在python2中filter直接返回列表，但在python3中返回迭代器
#        因此为了兼容python3, 需要list转换一下

# Output: [-5, -4, -3, -2, -1]
```

这个filter类似于一个for循环，但它是一个内置函数，并且更快。

注意：如果map和filter对你来说看起来并不优雅的话，那么你可以看看另外一章：列表/字典/元组推导式。

> 译者注：大部分情况下推导式的可读性更好

Reduce

Reduce

当需要对一个列表进行一些计算并返回结果时，Reduce 是个非常有用的函数。举个例子，当你需要计算一个整数列表的乘积时。

通常在 python 中你可能会使用基本的 for 循环来完成这个任务。

现在我们来试试 reduce：

```
from functools import reduce
product = reduce( (lambda x, y: x * y), [1, 2, 3, 4] )

# Output: 24
```

set 数据结构

set(集合)数据结构

set(集合)是一个非常有用的数据结构。它与列表(list)的行为类似，区别在于set不能包含重复的值。
这在很多情况下非常有用。例如你可能想检查列表中是否包含重复的元素，你有两个选择，第一个需要使用for循环，就像这样：

```
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']

duplicates = []
for value in some_list:
    if some_list.count(value) > 1:
        if value not in duplicates:
            duplicates.append(value)

print(duplicates)
### 输出: ['b', 'n']
```

但还有一种更简单更优雅的解决方案，那就是使用集合(sets)，你直接这样做：

```
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
duplicates = set([x for x in some_list if some_list.count(x) > 1])
print(duplicates)
### 输出: set(['b', 'n'])
```

集合还有一些其它方法，下面我们介绍其中一部分。

交集

你可以对比两个集合的交集（两个集合中都有的数据），如下：

```
valid = set(['yellow', 'red', 'blue', 'green', 'black'])
input_set = set(['red', 'brown'])
print(input_set.intersection(valid))
### 输出: set(['red'])
```

差集

你可以用差集(difference)找出无效的数据，相当于用一个集合减去另一个集合的数据，例如：

```
valid = set(['yellow', 'red', 'blue', 'green', 'black'])
input_set = set(['red', 'brown'])
print(input_set.difference(valid))
### 输出: set(['brown'])
```

你也可以用{}符号来创建集合，如：

```
a_set = {'red', 'blue', 'green'}
```

```
print(type(a_set))
### 输出: <type 'set'>
```

集合还有一些其它方法，我会建议访问官方文档并做个快速阅读。

三元运算符

三元运算符

三元运算符通常在Python里被称为条件表达式，这些表达式基于真(true)/假(false)的条件判断，在Python 2.4以上才有了三元操作。

下面是一个伪代码和例子：

伪代码:

```
#如果条件为真，返回真 否则返回假
condition_is_true if condition else condition_is_false
```

例子:

```
is_fat = True
state = "fat" if is_fat else "not fat"
```

它允许用简单的一行快速判断，而不是使用复杂的多行if语句。 这在大多数时候非常有用，而且可以使代码简单可维护。

另一个晦涩一点的用法比较少见，它使用了元组，请继续看：

伪代码:

```
#(返回假，返回真)[真或假]
(if_test_is_false, if_test_is_true)[test]
```

例子:

```
fat = True
fitness = ("skinny", "fat")[fat]
print("Ali is", fitness)
#输出: Ali is fat
```

这之所以能正常工作，是因为在Python中，True等于1，而False等于0，这就相当于在元组中使用0和1来选取数据。

上面的例子没有被广泛使用，而且Python玩家一般不喜欢那样，因为没有Python味儿(Pythonic)。这样的用法很容易把真正的数据与True/False弄混。

另外一个不使用元组条件表达式的缘故是因为在元组中会把两个条件都执行，而 `if-else` 的条件表达式不会这样。

例如:

```
condition = True
print(2 if condition else 1 / 0)
#输出: 2

print((1 / 0, 2)[condition])
```

```
#输出ZeroDivisionError异常
```

这是因为在元组中是先建数据，然后用True(1)/False(0)来索引到数据。 而`if-else`条件表达式遵循普通的`if-else`逻辑树， 因此，如果逻辑中的条件异常，或者是重计算型（计算较久）的情况下，最好尽量避免使用元组条件表达式。

装饰器

装饰器

装饰器(Decorators)是Python的一个重要部分。简单地说：他们是修改其他函数的功能的函数。他们有助于让我们的代码更简短，也更Pythonic（Python范儿）。大多数初学者不知道在哪儿使用它们，所以我将要分享下，哪些区域里装饰器可以让你的代码更简洁。

首先，让我们讨论下如何写你自己的装饰器。

这可能是最难掌握的概念之一。我们会每次只讨论一个步骤，这样你能完全理解它。

一切皆对象

一切皆对象

首先我们来理解下Python中的函数

```
def hi(name="yasoob"):
    return "hi " + name

print(hi())
# output: 'hi yasoob'

# 我们甚至可以将一个函数赋值给一个变量，比如
greet = hi
# 我们这里没有在使用小括号，因为我们并不是在调用hi函数
# 而是在将它放在greet变量里头。我们尝试运行下这个

print(greet())
# output: 'hi yasoob'

# 如果我们删掉旧的hi函数，看看会发生什么！
del hi
print(hi())
#outputs: NameError

print(greet())
#outputs: 'hi yasoob'
```

在函数中定义函数

在函数中定义函数

刚才那些就是函数的基本知识了。我们来让你的知识更进一步。在Python中我们可以在一个函数中定义另一个函数：

```
def hi(name="yasoob"):
    print("now you are inside the hi() function")

    def greet():
        return "now you are in the greet() function"

    def welcome():
        return "now you are in the welcome() function"

    print(greet())
    print(welcome())
    print("now you are back in the hi() function")

hi()
#output:now you are inside the hi() function
#       now you are in the greet() function
#       now you are in the welcome() function
#       now you are back in the hi() function

# 上面展示了无论何时你调用hi(), greet()和welcome()将会同时被调用。
# 然后greet()和welcome()函数在hi()函数之外是不能访问的，比如：

greet()
#outputs: NameError: name 'greet' is not defined
```

那现在我们知道了可以在函数中定义另外的函数。也就是说：我们可以创建嵌套的函数。现在你需要再多学一点，就是函数也能返回函数。

从函数中返回函数

从函数中返回函数

其实并不需要在一个函数里去执行另一个函数，我们也可以将其作为输出返回出来：

```
def hi(name="yasoob"):
    def greet():
        return "now you are in the greet() function"

    def welcome():
        return "now you are in the welcome() function"

    if name == "yasoob":
        return greet
    else:
        return welcome

a = hi()
print(a)
#outputs: <function greet at 0x7f2143c01500>

#上面清晰地展示了`a`现在指向到hi()函数中的greet()函数
#现在试试这个

print(a())
#outputs: now you are in the greet() function
```

再次看看这个代码。在`if/else`语句中我们返回`greet`和`welcome`，而不是`greet()`和`welcome()`。为什么那样？这是因为当你把一对小括号放在后面，这个函数就会执行；然而如果你不放括号在它后面，那它可以被到处传递，并且可以赋值给别的变量而不去执行它。

你明白了吗？让我再稍微多解释点细节。

当我们写下`a = hi()`，`hi()`会被执行，而由于`name`参数默认是*yasoob*，所以函数`greet`被返回了。如果我们把语句改为`a = hi(name = "ali")`，那么`welcome`函数将被返回。我们还可以打印出`hi()()`，这会输出*now you are in the greet() function*。

将函数作为参数传给另一个函数

将函数作为参数传给另一个函数

```
def hi():
    return "hi yasoob!"

def doSomethingBeforeHi(func):
    print("I am doing some boring work before executing hi()")
    print(func())

doSomethingBeforeHi(hi)
#outputs:I am doing some boring work before executing hi()
#        hi yasoob!
```

现在你已经具备所有必需知识，来进一步学习装饰器真正是什么了。装饰器让你在一个函数的前后去执行代码。

你的第一个装饰器

你的第一个装饰器

在上一个例子里，其实我们已经创建了一个装饰器！现在我们修改下上一个装饰器，并编写一个稍微更有用点的程序：

```
def a_new_decorator(a_func):

    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()'

        a_func()

        print("I am doing some boring work after executing a_func()"

    return wrapTheFunction

def a_function_requiring_decoration():
    print("I am the function which needs some decoration to remove my

a_function_requiring_decoration()
#outputs: "I am the function which needs some decoration to remove my

a_function_requiring_decoration = a_new_decorator(a_function_requirin
#now a_function_requiring_decoration is wrapped by wrapTheFunction()

a_function_requiring_decoration()
#outputs:I am doing some boring work before executing a_func()
#        I am the function which needs some decoration to remove my
#        I am doing some boring work after executing a_func()
```

你看明白了吗？我们刚刚应用了之前学习到的原理。这正是python中装饰器做的事情！它们封装一个函数，并且用这样或者那样的方式来修改它的行为。现在你也许疑惑，我们在代码里并没有使用@符号？那只是一个简短的方式来生成一个被装饰的函数。这里是我们如何使用@来运行之前的代码：

```
@a_new_decorator
def a_function_requiring_decoration():
    """Hey you! Decorate me!"""
    print("I am the function which needs some decoration to "
          "remove my foul smell")

a_function_requiring_decoration()
#outputs: I am doing some boring work before executing a_func()
#         I am the function which needs some decoration to remove my
#         I am doing some boring work after executing a_func()

#the @a_new_decorator is just a short way of saying:
a_function_requiring_decoration = a_new_decorator(a_function_requirin
```

希望你现在对Python装饰器的工作原理有一个基本的理解。如果我们运行如下代码会存在一个问题：

```
print(a_function_requiring_decoration.__name__)
# Output: wrapTheFunction
```

这并不是我们想要的！Ouput输出应该是“a_function_requiring_decoration”。这里的函数被warpTheFunction替代了。它重写了我们函数的名字和注释文档(docstring)。幸运的是Python提供给我们一个简单的函数来解决这个问题，那就是functools.wraps。我们修改上一个例子来使用functools.wraps：

```
from functools import wraps

def a_new_decorator(a_func):
    @wraps(a_func)
    def wrapTheFunction():
        print("I am doing some boring work before executing a_func()"
        a_func()
        print("I am doing some boring work after executing a_func()"
    return wrapTheFunction

@a_new_decorator
def a_function_requiring_decoration():
    """Hey yo! Decorate me!"""
    print("I am the function which needs some decoration to "
          "remove my foul smell")

print(a_function_requiring_decoration.__name__)
# Output: a_function_requiring_decoration
```

现在好多了。我们接下来学习装饰器的一些常用场景。

蓝本规范:

```
from functools import wraps
def decorator_name(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        if not can_run:
            return "Function will not run"
        return f(*args, **kwargs)
    return decorated

@decorator_name
def func():
    return("Function is running")

can_run = True
print(func())
# Output: Function is running

can_run = False
print(func())
# Output: Function will not run
```

注意：@wraps接受一个函数来进行装饰，并加入了复制函数名称、注释文档、参数列表等等的功能。这可以让我们在装饰器里面访问在装饰之前的函数的属性。

使用场景

使用场景

现在我们来看一下装饰器在哪些地方特别耀眼，以及使用它可以让一些事情管理起来变得更简单。

授权

授权(Authorization)

装饰器能有助于检查某个人是否被授权去使用一个web应用的端点(endpoint)。它们被大量使用于Flask和Django web框架中。这里是一个例子来使用基于装饰器的授权：

```
from functools import wraps

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            authenticate()
        return f(*args, **kwargs)
    return decorated
```

日志

日志(Logging)

日志是装饰器运用的另一个亮点。这是个例子：

```
from functools import wraps

def logit(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__ + " was called")
        return func(*args, **kwargs)
    return with_logging

@logit
def addition_func(x):
   """Do some math."""
   return x + x


result = addition_func(4)
# Output: addition_func was called
```

我敢肯定你已经在思考装饰器的一个其他聪明用法了。

带参数的装饰器

带参数的装饰器

来想想这个问题，难道@wraps不也是个装饰器吗？但是，它接收一个参数，就像任何普通的函数能做的那样。那么，为什么我们不也那样做呢？

这是因为，当你使用@my_decorator语法时，你是在应用一个以单个函数作为参数的一个包裹函数。记住，Python里每个东西都是一个对象，而且这包括函数！记住了这些，我们可以编写一下能返回一个包裹函数的函数。

在函数中嵌入装饰器

在函数中嵌入装饰器

我们回到日志的例子，并创建一个包裹函数，能让我们指定一个用于输出的日志文件。

```python
from functools import wraps

def logit(logfile='out.log'):
    def logging_decorator(func):
        @wraps(func)
        def wrapped_function(*args, **kwargs):
            log_string = func.__name__ + " was called"
            print(log_string)
            # 打开logfile，并写入内容
            with open(logfile, 'a') as opened_file:
                # 现在将日志打到指定的logfile
                opened_file.write(log_string + '\n')
            return func(*args, **kwargs)
        return wrapped_function
    return logging_decorator

@logit()
def myfunc1():
    pass

myfunc1()
# Output: myfunc1 was called
# 现在一个叫做 out.log 的文件出现了，里面的内容就是上面的字符串

@logit(logfile='func2.log')
def myfunc2():
    pass

myfunc2()
# Output: myfunc2 was called
# 现在一个叫做 func2.log 的文件出现了，里面的内容就是上面的字符串
```

装饰器类

装饰器类

现在我们有了能用于正式环境的logit装饰器，但当我们的应用的某些部分还比较脆弱时，异常也许是需要更紧急关注的事情。比方说有时你只想打日志到一个文件。而有时你想把引起你注意的问题发送到一个email，同时也保留日志，留个记录。这是一个使用继承的场景，但目前为止我们只看到过用来构建装饰器的函数。

幸运的是，类也可以用来构建装饰器。那我们现在以一个类而不是一个函数的方式，来重新构建logit。

```
from functools import wraps

class logit(object):
    def __init__(self, logfile='out.log'):
        self.logfile = logfile

    def __call__(self, func):
        @wraps(func)
        def wrapped_function(*args, **kwargs):
            log_string = func.__name__ + " was called"
            print(log_string)
            # 打开logfile并写入
            with open(self.logfile, 'a') as opened_file:
                # 现在将日志打到指定的文件
                opened_file.write(log_string + '\n')
            # 现在，发送一个通知
            self.notify()
            return func(*args, **kwargs)
        return wrapped_function

    def notify(self):
        # logit只打日志，不做别的
        pass
```

这个实现有一个附加优势，在于比嵌套函数的方式更加整洁，而且包裹一个函数还是使用跟以前一样的语法：

```
@logit()
def myfunc1():
    pass
```

现在，我们给logit创建子类，来添加email的功能(虽然email这个话题不会在这里展开)。

```
class email_logit(logit):
    '''
    一个logit的实现版本，可以在函数调用时发送email给管理员
    '''
    def __init__(self, email='admin@myproject.com', *args, **kwargs)
```

```
        self.email = email
        super(email_logit, self).__init__(*args, **kwargs)

    def notify(self):
        # 发送一封email到self.email
        # 这里就不做实现了
        pass
```

从现在起，@email_logit将会和@logit产生同样的效果，但是在打日志的基础上，还会多发送一封邮件给管理员。

Global和Return

Global和Return

你也许遇到过, python中一些函数在最尾部有一个return关键字。你知道它是干嘛吗？它和其他语言的return类似。我们来检查下这个小函数：

```
def add(value1, value2):
    return value1 + value2

result = add(3, 5)
print(result)
# Output: 8
```

上面这个函数将两个值作为输入，然后输出它们相加之和。我们也可以这样做：

```
def add(value1,value2):
    global result
    result = value1 + value2

add(3,5)
print(result)
# Output: 8
```

那首先我们来谈谈第一段也就是包含return关键字的代码。那个函数把值赋给了调用它的变量（也就是例子中的result变量）。
大多数境况下，你并不需要使用global关键字。然而我们也来检查下另外一段也就是包含global关键字的代码。 那个函数生成了一个global（全局）变量result。

global在这的意思是什么？global变量意味着我们可以在函数以外的区域都能访问这个变量。让我们通过一个例子来证明它：

```
# 首先，是没有使用global变量
def add(value1, value2):
    result = value1 + value2

add(2, 4)
print(result)

# Oh 糟了，我们遇到异常了。为什么会这样？
# python解释器报错说没有一个叫result的变量。
# 这是因为result变量只能在创建它的函数内部才允许访问，除非它是全局的(global)。
Traceback (most recent call last):
  File "", line 1, in
    result
NameError: name 'result' is not defined

# 现在我们运行相同的代码，不过是在将result变量设为global之后
def add(value1, value2):
    global result
```

```
    result = value1 + value2

add(2, 4)
print(result)
6
```

如我们所愿，在第二次运行时没有异常了。在实际的编程时，你应该试着避开global关键字，它只会让生活变得艰难，因为它引入了多余的变量到全局作用域了。

多个return值

多个return值

那如果你想从一个函数里返回两个变量而不是一个呢？
新手们有若干种方法。最著名的方法，是使用global关键字。让我们看下这个没用的例子：

```
def profile():
    global name
    global age
    name = "Danny"
    age = 30

profile()
print(name)
# Output: Danny

print(age)
# Output: 30
```

注意: 不要试着使用上述方法。重要的事情说三遍，不要试着使用上述方法！

有些人试着在函数结束时，返回一个包含多个值的tuple(元组)，list(列表)或者dict(字典),来解决这个问题。这是一种可行的方式，而且使用起来像一个黑魔法：

```
def profile():
    name = "Danny"
    age = 30
    return (name, age)

profile_data = profile()
print(profile_data[0])
# Output: Danny

print(profile_data[1])
# Output: 30
```

或者按照更常见的惯例：

```
def profile():
    name = "Danny"
    age = 30
    return name, age
```

这是一种比列表和字典更好的方式。不要使用global关键字，除非你知道你正在做什么。global也许在某些场景下是一个更好的选择（但其中大多数情况都不是）。

对象变动 Mutation

对象变动(Mutation)

Python中可变(**mutable**)与不可变(**immutable**)的数据类型让新手很是头痛。简单的说，可变(mutable)意味着"可以被改动"，而不可变(immutable)的意思是“常量(constant)”。想把脑筋转动起来吗？考虑下这个例子：

```
foo = ['hi']
print(foo)
# Output: ['hi']

bar = foo
bar += ['bye']
print(foo)
# Output: ['hi', 'bye']
```

刚刚发生了什么？我们预期的不是那样！我们期望看到是这样的：

```
foo = ['hi']
print(foo)
# Output: ['hi']

bar = foo
bar += ['bye']

print(foo)
# Output: ['hi']

print(bar)
# Output: ['hi', 'bye']
```

这不是一个bug。这是对象可变性(**mutability**)在作怪。每当你将一个变量赋值为另一个可变类型的变量时，对这个数据的任意改动会同时反映到这两个变量上去。新变量只不过是老变量的一个别名而已。这个情况只是针对可变数据类型。下面的函数和可变数据类型让你一下就明白了：

```
def add_to(num, target=[]):
    target.append(num)
    return target

add_to(1)
# Output: [1]

add_to(2)
# Output: [1, 2]

add_to(3)
# Output: [1, 2, 3]
```

你可能预期它表现的不是这样子。你可能希望，当你调用add_to时，有一个新的列表被创建，就像这样：

```
def add_to(num, target=[]):
    target.append(num)
    return target

add_to(1)
# Output: [1]

add_to(2)
# Output: [2]

add_to(3)
# Output: [3]
```

啊哈！这次又没有达到预期，是列表的可变性在作怪。在Python中当函数被定义时，默认参数只会运算一次，而不是每次被调用时都会重新运算。你应该永远不要定义可变类型的默认参数，除非你知道你正在做什么。你应该像这样做：

```
def add_to(element, target=None):
    if target is None:
        target = []
    target.append(element)
    return target
```

现在每当你在调用这个函数不传入target参数的时候，一个新的列表会被创建。举个例子：

```
add_to(42)
# Output: [42]

add_to(42)
# Output: [42]

add_to(42)
# Output: [42]
```

slots魔法

__slots__魔法

在Python中，每个类都有实例属性。默认情况下Python用一个字典来保存一个对象的实例属性。这非常有用，因为它允许我们在运行时去设置任意的新属性。

然而，对于有着已知属性的小类来说，它可能是个瓶颈。这个字典浪费了很多内存。Python不能在对象创建时直接分配一个固定量的内存来保存所有的属性。因此如果你创建许多对象（我指的是成千上万个），它会消耗掉很多内存。
不过还是有一个方法来规避这个问题。这个方法需要使用__slots__来告诉Python不要使用字典，而且只给一个固定集合的属性分配空间。

这里是一个使用与不使用__slots__的例子：

- 不使用 __slots__:

```
class MyClass(object):
  def __init__(self, name, identifier):
      self.name = name
      self.identifier = identifier
      self.set_up()
  # ...
```

- 使用 __slots__:

```
class MyClass(object):
  __slots__ = ['name', 'identifier']
  def __init__(self, name, identifier):
      self.name = name
      self.identifier = identifier
      self.set_up()
  # ...
```

第二段代码会为你的内存减轻负担。通过这个技巧，有些人已经看到内存占用率几乎40%~50%的减少。

稍微备注一下，你也许需要试一下PyPy。它已经默认地做了所有这些优化。

以下你可以看到一个例子，它用IPython来展示在有与没有__slots__情况下的精确内存占用，感谢 https://github.com/ianozsvald/ipython_memory_usage

```
Python 3.4.3 (default, Jun  6 2015, 13:32:34)
Type "copyright", "credits" or "license" for more information.

IPython 4.0.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra detail:
```

```
In [1]: import ipython_memory_usage.ipython_memory_usage as imu

In [2]: imu.start_watching_memory()
In [2] used 0.0000 MiB RAM in 5.31s, peaked 0.00 MiB above current,

In [3]: %cat slots.py
class MyClass(object):
        __slots__ = ['name', 'identifier']
        def __init__(self, name, identifier):
                self.name = name
                self.identifier = identifier

num = 1024*256
x = [MyClass(1,1) for i in range(num)]
In [3] used 0.2305 MiB RAM in 0.12s, peaked 0.00 MiB above current,

In [4]: from slots import *
In [4] used 9.3008 MiB RAM in 0.72s, peaked 0.00 MiB above current,

In [5]: %cat noslots.py
class MyClass(object):
        def __init__(self, name, identifier):
                self.name = name
                self.identifier = identifier

num = 1024*256
x = [MyClass(1,1) for i in range(num)]
In [5] used 0.1758 MiB RAM in 0.12s, peaked 0.00 MiB above current,

In [6]: from noslots import *
In [6] used 22.6680 MiB RAM in 0.80s, peaked 0.00 MiB above current,
```

虚拟环境

虚拟环境(virtualenv)

你听说过virtualenv吗？

如果你是一位初学者，你可能没有听说过virtualenv；但如果你是位经验丰富的程序员，那么它可能是你的工具集的重要组织部分。

那么，什么是virtualenv?

Virtualenv 是一个工具，它能够帮我们创建一个独立(隔离)的Python环境。想象你有一个应用程序，依赖于版本为2的第三方模块，但另一个程序依赖的版本是3，请问你如何使用和开发这些应用程序？

如果你把一切都安装到了/usr/lib/python2.7/site-packages（或者其它平台的标准位置），那很容易出现某个模块被升级而你却不知道的情况。

在另一种情况下，想象你有一个已经开发完成的程序，但是你不想更新它所依赖的第三方模块版本；但你已经开始另一个程序，需要这些第三方模块的版本。

用什么方式解决？

使用virtualenv！针对每个程序创建独立（隔离）的Python环境，而不是在全局安装所依赖的模块。

要安装它，只需要在命令行中输入以下命令：

```
$ pip install virtualenv
```

最重要的命令是：

```
$ virtualenv myproject
$ source myproject/bin/activate
```

执行第一个命令在myproject文件夹创建一个隔离的virtualenv环境，第二个命令激活这个隔离的环境(virtualenv)。

在创建virtualenv时，你必须做出决定：这个virtualenv是使用系统全局的模块呢？还是只使用这个virtualenv内的模块。
默认情况下，virtualenv不会使用系统全局模块。

如果你想让你的virtualenv使用系统全局模块，请使用--system-site-packages参数创建你的virtualenv，例如：

```
virtualenv --system-site-packages mycoolproject
```

使用以下命令可以退出这个virtualenv:

```
$ deactivate
```

运行之后将恢复使用你系统全局的Python模块。

福利

你可以使用smartcd来帮助你管理你的环境，当你切换目录时，它可以帮助你激活（activate）和退出（deactivate）你的virtualenv。我已经用了很多次，很喜欢它。你可以在github(https://github.com/cxreg/smartcd) 上找到更多关于它的资料。

这只是一个virtualenv的简短介绍，你可以在 http://docs.python-guide.org/en/latest/dev/virtualenvs/ 找到更多信息。

容器 Collections

容器(Collections)

Python附带一个模块，它包含许多容器数据类型，名字叫作`collections`。我们将讨论它的作用和用法。

我们将讨论的是：

- defaultdict
- counter
- deque
- namedtuple
- enum.Enum (包含在Python 3.4以上)

defaultdict

我个人使用`defaultdict`较多，与`dict`类型不同，你不需要检查**key**是否存在，所以我们能这样做：

```
from collections import defaultdict

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

favourite_colours = defaultdict(list)

for name, colour in colours:
    favourite_colours[name].append(colour)

print(favourite_colours)
```

运行输出

```
# defaultdict(<type 'list'>,
#    {'Arham': ['Green'],
#     'Yasoob': ['Yellow', 'Red'],
#     'Ahmed': ['Silver'],
#     'Ali': ['Blue', 'Black']
# })
```

另一种重要的是例子就是：当你在一个字典中对一个键进行嵌套赋值时，如果这个键不存

在，会触发keyError异常。 defaultdict允许我们用一个聪明的方式绕过这个问题。 首先我分享一个使用dict触发KeyError的例子，然后提供一个使用defaultdict的解决方案。

问题：

```
some_dict = {}
some_dict['colours']['favourite'] = "yellow"

## 异常输出：KeyError: 'colours'
```

解决方案：

```
import collections
tree = lambda: collections.defaultdict(tree)
some_dict = tree()
some_dict['colours']['favourite'] = "yellow"

## 运行正常
```

你可以用json.dumps打印出some_dict，例如：

```
import json
print(json.dumps(some_dict))

## 输出: {"colours": {"favourite": "yellow"}}
```

counter

Counter是一个计数器，它可以帮助我们针对某项数据进行计数。比如它可以用来计算每个人喜欢多少种颜色：

```
from collections import Counter

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

favs = Counter(name for name, colour in colours)
print(favs)

## 输出:
## Counter({
##     'Yasoob': 2,
##     'Ali': 2,
##     'Arham': 1,
##     'Ahmed': 1
##  })
```

我们也可以在利用它统计一个文件，例如：

```
with open('filename', 'rb') as f:
    line_count = Counter(f)
print(line_count)
```

deque

deque提供了一个双端队列，你可以从头/尾两端添加或删除元素。要想使用它，首先我们要从collections中导入deque模块：

```
from collections import deque
```

现在，你可以创建一个deque对象。

```
d = deque()
```

它的用法就像python的list，并且提供了类似的方法，例如：

```
d = deque()
d.append('1')
d.append('2')
d.append('3')

print(len(d))

## 输出: 3

print(d[0])

## 输出: '1'

print(d[-1])

## 输出: '3'
```

你可以从两端取出(pop)数据：

```
d = deque(range(5))
print(len(d))

## 输出: 5

d.popleft()

## 输出: 0

d.pop()

## 输出: 4

print(d)
```

```
## 输出: deque([1, 2, 3])
```

我们也可以限制这个列表的大小，当超出你设定的限制时，数据会从对队列另一端被挤出去(pop)。
最好的解释是给出一个例子：

```
d = deque(maxlen=30)
```

现在当你插入30条数据时，最左边一端的数据将从队列中删除。

你还可以从任一端扩展这个队列中的数据：

```
d = deque([1,2,3,4,5])
d.extendleft([0])
d.extend([6,7,8])
print(d)

## 输出: deque([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

namedtuple

您可能已经熟悉元组。
一个元组是一个不可变的列表，你可以存储一个数据的序列，它和命名元组(`namedtuples`)非常像，但有几个关键的不同。
主要相似点是都不像列表，你不能修改元组中的数据。为了获取元组中的数据，你需要使用整数作为索引：

```
man = ('Ali', 30)
print(man[0])

## 输出: Ali
```

嗯，那`namedtuples`是什么呢？它把元组变成一个针对简单任务的容器。你不必使用整数索引来访问一个`namedtuples`的数据。你可以像字典(`dict`)一样访问`namedtuples`，但`namedtuples`是不可变的。

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")

print(perry)

## 输出: Animal(name='perry', age=31, type='cat')

print(perry.name)

## 输出: 'perry'
```

现在你可以看到，我们可以用名字来访问`namedtuple`中的数据。我们再继续分析它。一个命名元组(`namedtuple`)有两个必需的参数。它们是元组名称和字段名称。

在上面的例子中，我们的元组名称是`Animal`，字段名称是'name'，'age'和'type'。

namedtuple让你的元组变得自文档了。你只要看一眼就很容易理解代码是做什么的。
你也不必使用整数索引来访问一个命名元组，这让你的代码更易于维护。
而且，**namedtuple**的每个实例没有对象字典，所以它们很轻量，与普通的元组比，并不需要更多的内存。这使得它们比字典更快。

然而，要记住它是一个元组，属性值在namedtuple中是不可变的，所以下面的代码不能工作：

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")
perry.age = 42

## 输出:
## Traceback (most recent call last):
##     File "", line 1, in
## AttributeError: can't set attribute
```

你应该使用命名元组来让代码自文档，它们向后兼容于普通的元组，这意味着你可以既使用整数索引，也可以使用名称来访问namedtuple：

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")
print(perry[0])

## 输出: perry
```

最后，你可以将一个命名元组转换为字典，方法如下：

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31, type="cat")
print(perry._asdict())

## 输出: OrderedDict([('name', 'Perry'), ('age', 31), ...
```

enum.Enum (Python 3.4+)

另一个有用的容器是枚举对象，它属于enum模块，存在于Python 3.4以上版本中（同时作为一个独立的PyPI包enum34供老版本使用）。Enums(枚举类型)基本上是一种组织各种东西的方式。

让我们回顾一下上一个'Animal'命名元组的例子。
它有一个type字段，问题是，type是一个字符串。
那么问题来了，万一程序员输入了Cat，因为他按到了Shift键，或者输入了'CAT'，甚至'kitten'？

枚举可以帮助我们避免这个问题，通过不使用字符串。考虑以下这个例子：

```
from collections import namedtuple
from enum import Enum

class Species(Enum):
    cat = 1
    dog = 2
    horse = 3
    aardvark = 4
    butterfly = 5
    owl = 6
    platypus = 7
    dragon = 8
    unicorn = 9
    # 依次类推

    # 但我们并不想关心同一物种的年龄，所以我们可以使用一个别名
    kitten = 1  # (译者注：幼小的猫咪)
    puppy = 2   # (译者注：幼小的狗狗)

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31, type=Species.cat)
drogon = Animal(name="Drogon", age=4, type=Species.dragon)
tom = Animal(name="Tom", age=75, type=Species.cat)
charlie = Animal(name="Charlie", age=2, type=Species.kitten)
```

现在，我们进行一些测试：

```
>>> charlie.type == tom.type
True
>>> charlie.type
<Species.cat: 1>
```

这样就没那么容易错误，我们必须更明确，而且我们应该只使用定义后的枚举类型。

有三种方法访问枚举数据，例如以下方法都可以获取到'cat'的值：

```
Species(1)
Species['cat']
Species.cat
```

这只是一个快速浏览collections模块的介绍，建议你阅读本文最后的官方文档。

枚举 Enumerate

枚举

枚举(enumerate)是Python内置函数。它的用处很难在简单的一行中说明，但是大多数的新人，甚至一些高级程序员都没有意识到它。

它允许我们遍历数据并自动计数，

下面是一个例子：

```
for counter, value in enumerate(some_list):
    print(counter, value)
```

不只如此，enumerate也接受一些可选参数，这使它更有用。

```
my_list = ['apple', 'banana', 'grapes', 'pear']
for c, value in enumerate(my_list, 1):
    print(c, value)

# 输出:
(1, 'apple')
(2, 'banana')
(3, 'grapes')
(4, 'pear')
```

上面这个可选参数允许我们定制从哪个数字开始枚举。
你还可以用来创建包含索引的元组列表， 例如：

```
my_list = ['apple', 'banana', 'grapes', 'pear']
counter_list = list(enumerate(my_list, 1))
print(counter_list)
# 输出: [(1, 'apple'), (2, 'banana'), (3, 'grapes'), (4, 'pear')]
```

对象自省

对象自省

自省(introspection)，在计算机编程领域里，是指在运行时来判断一个对象的类型的能力。它是Python的强项之一。Python中所有一切都是一个对象，而且我们可以仔细勘察那些对象。Python还包含了许多内置函数和模块来帮助我们。

dir

dir

在这个小节里我们会学习到dir以及它在自省方面如何给我们提供便利。

它是用于自省的最重要的函数之一。它返回一个列表，列出了一个对象所拥有的属性和方法。这里是一个例子：

```
my_list = [1, 2, 3]
dir(my_list)
# Output: ['__add__', '__class__', '__contains__', '__delattr__', '_
# '__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__ge
# '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '_
# '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
# '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed_
# '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str_
# '__subclasshook__', 'append', 'count', 'extend', 'index', 'insert',
# 'remove', 'reverse', 'sort']
```

上面的自省给了我们一个列表对象的所有方法的名字。当你没法回忆起一个方法的名字，这会非常有帮助。如果我们运行dir()而不传入参数，那么它会返回当前作用域的所有名字。

type和id

type和id

type函数返回一个对象的类型。举个例子：

```
print(type(''))
# Output: <type 'str'>

print(type([]))
# Output: <type 'list'>

print(type({}))
# Output: <type 'dict'>

print(type(dict))
# Output: <type 'type'>

print(type(3))
# Output: <type 'int'>
```

id()函数返回任意不同种类对象的唯一ID，举个例子：

```
name = "Yasoob"
print(id(name))
# Output: 139972439030304
```

inspect模块

inspect模块

inspect模块也提供了许多有用的函数，来获取活跃对象的信息。比方说，你可以查看一个对象的成员，只需运行：

```
import inspect
print(inspect.getmembers(str))
# Output: [('__add__', <slot wrapper '__add__' of ... ...
```

还有好多个其他方法也能有助于自省。如果你愿意，你可以去探索它们。

推导式 Comprehension

各种推导式(comprehensions)

推导式（又称解析式）是Python的一种独有特性，如果我被迫离开了它，我会非常想念。推导式是可以从一个数据序列构建另一个新的数据序列的结构体。 共有三种推导，在Python2和3中都有支持：

- 列表(`list`)推导式
- 字典(`dict`)推导式
- 集合(`set`)推导式

我们将一一进行讨论。一旦你知道了使用列表推导式的诀窍，你就能轻易使用任意一种推导式了。

列表推导式

列表推导式（list comprehensions）

列表推导式（又称列表解析式）提供了一种简明扼要的方法来创建列表。
它的结构是在一个中括号里包含一个表达式，然后是一个for语句，然后是0个或多个for或者if语句。那个表达式可以是任意的，意思是你可以在列表中放入任意类型的对象。返回结果将是一个新的列表，在这个以if和for语句为上下文的表达式运行完成之后产生。

规范

```
variable = [out_exp for out_exp in input_list if out_exp == 2]
```

这里是另外一个简明例子:

```
multiples = [i for i in range(30) if i % 3 is 0]
print(multiples)
# Output: [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

这将对快速生成列表非常有用。
有些人甚至更喜欢使用它而不是filter函数。
列表推导式在有些情况下超赞，特别是当你需要使用for循环来生成一个新列表。举个例子，你通常会这样做：

```
squared = []
for x in range(10):
    squared.append(x**2)
```

你可以使用列表推导式来简化它，就像这样：

```
squared = [x**2 for x in range(10)]
```

字典推导式

字典推导式（`dict` comprehensions）

字典推导和列表推导的使用方法是类似的。这里有个我最近发现的例子：

```
mcase = {'a': 10, 'b': 34, 'A': 7, 'Z': 3}

mcase_frequency = {
    k.lower(): mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0)
    for k in mcase.keys()
}

# mcase_frequency == {'a': 17, 'z': 3, 'b': 34}
```

在上面的例子中我们把同一个字母但不同大小写的值合并起来了。

就我个人来说没有大量使用字典推导式。

你还可以快速对换一个字典的键和值：

```
{v: k for k, v in some_dict.items()}
```

集合推导式

集合推导式（`set` comprehensions）

它们跟列表推导式也是类似的。 唯一的区别在于它们使用大括号{}。 举个例子：

```
squared = {x**2 for x in [1, 1, 2]}
print(squared)
# Output: {1, 4}
```

异常

异常

异常处理是一种艺术，一旦你掌握，会授予你无穷的力量。我将要向你展示我们能处理异常的一些方式。

最基本的术语里我们知道了try/except从句。可能触发异常产生的代码会放到try语句块里，而处理异常的代码会在except语句块里实现。这是一个简单的例子：

```
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('An IOError occurred. {}'.format(e.args[-1]))
```

上面的例子里，我们仅仅在处理一个IOError的异常。大部分初学者还不知道的是，我们可以处理多个异常。

处理多个异常

处理多个异常

我们可以使用三种方法来处理多个异常。

第一种方法需要把所有可能发生的异常放到一个元组里。像这样：

```
try:
    file = open('test.txt', 'rb')
except (IOError, EOFError) as e:
    print("An error occurred. {}".format(e.args[-1]))
```

另外一种方式是对每个单独的异常在单独的except语句块中处理。我们想要多少个except语句块都可以。这里是个例子：

```
try:
    file = open('test.txt', 'rb')
except EOFError as e:
    print("An EOF error occurred.")
    raise e
except IOError as e:
    print("An error occurred.")
    raise e
```

上面这个方式中，如果异常没有被第一个except语句块处理，那么它也许被下一个语句块处理，或者根本不会被处理。

现在，最后一种方式会捕获所有异常：

```
try:
    file = open('test.txt', 'rb')
except Exception:
    # 打印一些异常日志，如果你想要的话
    raise
```

当你不知道你的程序会抛出什么样的异常时，上面的方式可能非常有帮助。

finally从句

finally从句

我们把我们的主程序代码包裹进了try从句。然后我们把一些代码包裹进一个except从句，它会在try从句中的代码触发异常时执行。

在下面的例子中，我们还会使用第三个从句，那就是finally从句。包裹到finally从句中的代码不管异常是否触发都将会被执行。这可以被用来在脚本执行之后做清理工作。这里是个简单的例子：

```
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('An IOError occurred. {}'.format(e.args[-1]))
finally:
    print("This would be printed whether or not an exception occurre

# Output: An IOError occurred. No such file or directory
# This would be printed whether or not an exception occurred!
```

try/else从句

try/else从句

我们常常想在没有触发异常的时候执行一些代码。这可以很轻松地通过一个else从句来达到。

有人也许问了：如果你只是想让一些代码在没有触发异常的情况下执行，为啥你不直接把代码放在try里面呢？
回答是，那样的话这段代码中的任意异常都还是会被try捕获，而你并不一定想要那样。

大多数人并不使用else从句，而且坦率地讲我自己也没有大范围使用。这里是个例子：

```
try:
    print('I am sure no exception is going to occur!')
except Exception:
    print('exception')
else:
    # 这里的代码只会在try语句里没有触发异常时运行,
    # 但是这里的异常将 *不会* 被捕获
    print('This would only run if no exception occurs. And an error h
          'would NOT be caught.')
finally:
    print('This would be printed in every case.')

# Output: I am sure no exception is going to occur!
# This would only run if no exception occurs.
# This would be printed in every case.
```

else从句只会在没有异常的情况下执行，而且它会在finally语句之前执行。

lambda表达式

17. lambda表达式

lambda表达式是一行函数。
它们在其他语言中也被称为匿名函数。如果你不想在程序中对一个函数使用两次，你也许会想用lambda表达式，它们和普通的函数完全一样。

原型

```
lambda 参数:操作(参数)
```

例子

```
add = lambda x, y: x + y

print(add(3, 5))
# Output: 8
```

这还有一些lambda表达式的应用案例，可以在一些特殊情况下使用：

列表排序

```
a = [(1, 2), (4, 1), (9, 10), (13, -3)]
a.sort(key=lambda x: x[1])

print(a)
# Output: [(13, -3), (4, 1), (1, 2), (9, 10)]
```

列表并行排序

```
data = zip(list1, list2)
data = sorted(data)
list1, list2 = map(lambda t: list(t), zip(*data))
```

一行式

18. 一行式

本章节,我将向大家展示一些一行式的Python命令，这些程序将对你非常有帮助。

简易**Web Server**

你是否想过通过网络快速共享文件？好消息，Python为你提供了这样的功能。进入到你要共享文件的目录下并在命令行中运行下面的代码：

```
# Python 2
python -m SimpleHTTPServer

# Python 3
python -m http.server
```

漂亮的打印

你可以在Python REPL漂亮的打印出列表和字典。这里是相关的代码：

```
from pprint import pprint

my_dict = {'name': 'Yasoob', 'age': 'undefined', 'personality':
pprint(my_dict)
```

这种方法在字典上更为有效。此外，如果你想快速漂亮的从文件打印出json数据，那么你可以这么做：

```
cat file.json | python -m json.tool
```

脚本性能分析 这可能在定位你的脚本中的性能瓶颈时，会非常奏效：

```
python -m cProfile my_script.py
```

备注：`cProfile`是一个比`profile`更快的实现，因为它是用c写的

CSV转换为**json**

在命令行执行这条指令

```
python -c "import csv,json;print json.dumps(list(csv.reader(open
```

确保更换`csv_file.csv`为你想要转换的csv文件

列表辗平

您可以通过使用`itertools`包中的`itertools.chain.from_iterable`轻松快速的辗平一个列表。下面是一个简单的例子：

```
a_list = [[1, 2], [3, 4], [5, 6]]
print(list(itertools.chain.from_iterable(a_list)))
```

一行式

```
# Output: [1, 2, 3, 4, 5, 6]

# or
print(list(itertools.chain(*a_list)))
# Output: [1, 2, 3, 4, 5, 6]
```

一行式的构造器

避免类初始化时大量重复的赋值语句

```
class A(object):
    def __init__(self, a, b, c, d, e, f):
        self.__dict__.update({k: v for k, v in locals().items() i
```

更多的一行方法请参考Python官方文档。

For - Else

For - Else

循环是任何语言的一个必备要素。同样地，for循环就是Python的一个重要组成部分。然而还有一些东西是初学者并不知道的。我们将一个个讨论一下。

我们先从已经知道的开始。我们知道可以像这样使用for循环：

```
fruits = ['apple', 'banana', 'mango']
for fruit in fruits:
    print(fruit.capitalize())

# Output: Apple
#         Banana
#         Mango
```

这是一个for循环非常基础的结构。现在我们继续看看，Python的for循环的一些鲜为人所知的特性。

else语句

else从句

for循环还有一个else从句，我们大多数人并不熟悉。这个else从句会在循环正常结束时执行。这意味着，循环没有遇到任何break. 一旦你掌握了何时何地使用它，它真的会非常有用。我自己对它真是相见恨晚。

有个常见的构造是跑一个循环，并查找一个元素。如果这个元素被找到了，我们使用break来中断这个循环。有两个场景会让循环停下来。

- 第一个是当一个元素被找到，break被触发。
- 第二个场景是循环结束。

现在我们也许想知道其中哪一个，才是导致循环完成的原因。一个方法是先设置一个标记，然后在循环结束时打上标记。另一个是使用else从句。

这就是for/else循环的基本结构：

```
for item in container:
    if search_something(item):
        # Found it!
        process(item)
        break
else:
    # Didn't find anything..
    not_found_in_container()
```

考虑下这个简单的案例，它是我从官方文档里拿来的：

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n / x)
            break
```

它会找出2到10之间的数字的因子。现在是趣味环节了。我们可以加上一个附加的else语句块，来抓住质数，并且告诉我们：

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n / x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

else语句

使用C扩展

使用C扩展

CPython还为开发者实现了一个有趣的特性，使用Python可以轻松调用C代码

开发者有三种方法可以在自己的Python代码中来调用C编写的函数-`ctypes`，`SWIG`，`Python/C API`。每种方式也都有各自的利弊。

首先，我们要明确为什么要在Python中调用C？

常见原因如下：

- 你要提升代码的运行速度，而且你知道C要比Python快50倍以上
- C语言中有很多传统类库，而且有些正是你想要的，但你又不想用Python去重写它们
- 想对从内存到文件接口这样的底层资源进行访问
- 不需要理由，就是想这样做

CTypes

CTypes

Python中的ctypes模块可能是Python调用C方法中最简单的一种。ctypes模块提供了和C语言兼容的数据类型和函数来加载dll文件，因此在调用时不需对源文件做任何的修改。也正是如此奠定了这种方法的简单性。

示例如下

实现两数求和的C代码，保存为add.c

```
//sample C file to add 2 numbers - int and floats

#include <stdio.h>

int add_int(int, int);
float add_float(float, float);

int add_int(int num1, int num2){
    return num1 + num2;

}

float add_float(float num1, float num2){
    return num1 + num2;

}
```

接下来将C文件编译为.so文件(windows下为DLL)。下面操作会生成adder.so文件

```
#For Linux
$  gcc -shared -Wl,-soname,adder -o adder.so -fPIC add.c

#For Mac
$ gcc -shared -Wl,-install_name,adder.so -o adder.so -fPIC add.c
```

现在在你的Python代码中来调用它

```
from ctypes import *

#load the shared object file
adder = CDLL('./adder.so')

#Find sum of integers
res_int = adder.add_int(4,5)
print "Sum of 4 and 5 = " + str(res_int)

#Find sum of floats
a = c_float(5.5)
b = c_float(4.1)
```

```
add_float = adder.add_float
add_float.restype = c_float
print "Sum of 5.5 and 4.1 = ", str(add_float(a, b))
```

输出如下

```
Sum of 4 and 5 = 9
Sum of 5.5 and 4.1 =  9.60000038147
```

在这个例子中，C文件是自解释的，它包含两个函数，分别实现了整形求和和浮点型求和。

在Python文件中，一开始先导入ctypes模块，然后使用CDLL函数来加载我们创建的库文件。这样我们就可以通过变量`adder`来使用C类库中的函数了。当`adder.add_int()`被调用时，内部将发起一个对C函数`add_int`的调用。ctypes接口允许我们在调用C函数时使用原生Python中默认的字符串型和整型。

而对于其他类似布尔型和浮点型这样的类型，必须要使用正确的ctype类型才可以。如向`adder.add_float()`函数传参时, 我们要先将Python中的十进制值转化为c_float类型，然后才能传送给C函数。这种方法虽然简单，清晰，但是却很受限。例如，并不能在C中对对象进行操作。

SWIG

SWIG

SWIG是Simplified Wrapper and Interface Generator的缩写。是Python中调用C代码的另一种方法。在这个方法中，开发人员必须编写一个额外的接口文件来作为SWIG(终端工具)的入口。

Python开发者一般不会采用这种方法，因为大多数情况它会带来不必要的复杂。而当你有一个C/C++代码库需要被多种语言调用时，这将是个非常不错的选择。

示例如下(来自[SWIG官网](#))

~~~
```C
#include <time.h>
double My_variable = 3.0;

int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);

}

int my_mod(int x, int y) {
    return (x%y);

}

char *get_time()
{
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);

}
~~~

编译它

```
unix % swig -python example.i
unix % gcc -c example.c example_wrap.c \
    -I/usr/local/include/python2.1
unix % ld -shared example.o example_wrap.o -o _example.so
```

最后，Python的输出

```
>>> import example
>>> example.fact(5)
120
>>> example.my_mod(7,3)
1
>>> example.get_time()
'Sun Feb 11 23:01:07 1996'
```

```
>>>
```

我们可以看到，使用SWIG确实达到了同样的效果，虽然下了更多的工夫，但如果你的目标是多语言还是很值得的。

Python/C API

Python/C API

Python/C API可能是被最广泛使用的方法。它不仅简单，而且可以在C代码中操作你的Python对象。

这种方法需要以特定的方式来编写C代码以供Python去调用它。所有的Python对象都被表示为一种叫做PyObject的结构体，并且`Python.h`头文件中提供了各种操作它的函数。例如，如果PyObject表示为PyListType(列表类型)时，那么我们便可以使用`PyList_Size()`函数来获取该结构的长度，类似Python中的`len(list)`函数。大部分对Python原生对象的基础函数和操作在`Python.h`头文件中都能找到。

示例

编写一个C扩展，添加所有元素到一个Python列表(所有元素都是数字)

来看一下我们要实现的效果，这里演示了用Python调用C扩展的代码

```
#Though it looks like an ordinary python import, the addList module i
import addList

l = [1,2,3,4,5]
print "Sum of List - " + str(l) + " = " +  str(addList.add(l))
```

上面的代码和普通的Python文件并没有什么分别，导入并使用了另一个叫做`addList`的Python模块。唯一差别就是这个模块并不是用Python编写的，而是C。

接下来我们看看如何用C编写`addList`模块，这可能看起来有点让人难以接受，但是一旦你了解了这之中的各种组成，你就可以一往无前了。

```
//Python.h has all the required function definitions to manipulate tl
#include <Python.h>

//This is the function that is called from your python code
static PyObject* addList_add(PyObject* self, PyObject* args){

    PyObject * listObj;

    //The input arguments come as a tuple, we parse the args to get
    //In this case it's only one list variable, which will now be re
    if (! PyArg_ParseTuple( args, "O", &listObj ))
        return NULL;

    //length of the list
    long length = PyList_Size(listObj);

    //iterate over all the elements
    int i, sum =0;
    for (i = 0; i < length; i++) {
```

```
        //get an element out of the list - the element is also a pytl
        PyObject* temp = PyList_GetItem(listObj, i);
        //we know that object represents an integer - so convert it :
        long elem = PyInt_AsLong(temp);
        sum += elem;
    }

    //value returned back to python code - another python object
    //build value here converts the C long to a python integer
    return Py_BuildValue("i", sum);

}

//This is the docstring that corresponds to our 'add' function.
static char addList_docs[] =
"add(  ): add all elements of the list\n";

/* This table contains the relavent info mapping -
   <function-name in python module>, <actual-function>,
   <type-of-args the function expects>, <docstring associated with tl
 */
static PyMethodDef addList_funcs[] = {
    {"add", (PyCFunction)addList_add, METH_VARARGS, addList_docs},
    {NULL, NULL, 0, NULL}

};

/*
   addList is the module name, and this is the initialization block (
   <desired module name>, <the-info-table>, <module's-docstring>
 */
PyMODINIT_FUNC initaddList(void){
    Py_InitModule3("addList", addList_funcs,
            "Add all ze lists");

}
```

逐步解释

- Python.h头文件中包含了所有需要的类型(Python对象类型的表示)和函数定义(对Python对象的操作)
- 接下来我们编写将要在Python调用的函数, 函数传统的命名方式由{模块名}_{函数名}组成，所以我们将其命名为addList_add
- 然后填写想在模块内实现函数的相关信息表，每行一个函数，以空行作为结束
- 最后的模块初始化块签名为PyMODINIT_FUNC init{模块名}。

函数addList_add接受的参数类型为PyObject类型结构(同时也表示为元组类型，因为Python中万物皆为对象，所以我们先用PyObject来定义)。传入的参数则通过PyArg_ParseTuple()来解析。第一个参数是被解析的参数变量。第二个参数是一个字符串，告诉我们如何去解析元组中每一个元素。字符串的第n个字母正是代表着元组中第n个参数的类型。例如，"i"代表整形，"s"代表字符串类型, "O"则代表一个Python对象。接下来的参数都是你想要通过PyArg_ParseTuple()函数解析并保存的元素。这样参数的数量和模块中函数期待得到的参数数量就可以保持一致，并保证了位置的完整性。例如，我们想传入一个字符串，一个整数

和一个Python列表，可以这样去写

```
int n;
char *s;
PyObject* list;
PyArg_ParseTuple(args, "siO", &n, &s, &list);
```

在这种情况下，我们只需要提取一个列表对象，并将它存储在listObj变量中。然后用列表对象中的PyList_Size()函数来获取它的长度。就像Python中调用len(list)。

现在我们通过循环列表，使用PyList_GetItem(list, index)函数来获取每个元素。这将返回一个PyObject*对象。既然Python对象也能表示PyIntType，我们只要使用PyInt_AsLong(PyObj *)函数便可获得我们所需要的值。我们对每个元素都这样处理，最后再得到它们的总和。

总和将被转化为一个Python对象并通过Py_BuildValue()返回给Python代码，这里的i表示我们要返回一个Python整形对象。

现在我们已经编写完C模块了。将下列代码保存为setup.py

```
#build the modules

from distutils.core import setup, Extension

setup(name='addList', version='1.0',  \
      ext_modules=[Extension('addList', ['adder.c'])])
```

并且运行

```
python setup.py install
```

现在应该已经将我们的C文件编译安装到我们的Python模块中了。

在一番辛苦后，让我们来验证下我们的模块是否有效

```
#module that talks to the C code
import addList

l = [1,2,3,4,5]
print "Sum of List - " + str(l) + " = " +  str(addList.add(l))
```

输出结果如下

```
Sum of List - [1, 2, 3, 4, 5] = 15
```

如你所见，我们已经使用Python.h API成功开发出了我们第一个Python C扩展。这种方法看似复杂，但你一旦习惯，它将变的非常有效。

Python调用C代码的另一种方式便是使用Cython让Python编译的更快。但是Cython和传统的Python比起来可以将它理解为另一种语言，所以我们就不在这里过多描述了。

open函数

open函数

open 函数可以打开一个文件。超级简单吧？大多数时候，我们看到它这样被使用：

```
f = open('photo.jpg', 'r+')
jpgdata = f.read()
f.close()
```

我现在写这篇文章的原因，是大部分时间我看到open被这样使用。有三个错误存在于上面的代码中。你能把它们全指出来吗？如不能，请读下去。在这篇文章的结尾，你会知道上面的代码错在哪里，而且，更重要的是，你能在自己的代码里避免这些错误。现在我们从基础开始：

open的返回值是一个文件句柄，从操作系统托付给你的Python程序。一旦你处理完文件，你会想要归还这个文件句柄，只有这样你的程序不会超出一次能打开的文件句柄的数量上限。

显式地调用close关闭了这个文件句柄，但前提是只有在read成功的情况下。如果有任意异常正好在f = open(...)之后产生，f.close()将不会被调用（取决于Python解释器的做法，文件句柄可能还是会被归还，但那是另外的话题了）。为了确保不管异常是否触发，文件都能关闭，我们将其包裹成一个with语句:

```
with open('photo.jpg', 'r+') as f:
    jpgdata = f.read()
```

open的第一个参数是文件名。第二个(mode 打开模式)决定了这个文件如何被打开。

- 如果你想读取文件，传入r
- 如果你想读取并写入文件，传入r+
- 如果你想覆盖写入文件，传入w
- 如果你想在文件末尾附加内容，传入a

虽然有若干个其他的有效的mode字符串，但有可能你将永远不会使用它们。mode很重要，不仅因为它改变了行为，而且它可能导致权限错误。举个例子，我们要是在一个写保护的目录里打开一个jpg文件， open(.., 'r+')就失败了。mode可能包含一个扩展字符；让我们还可以以二进制方式打开文件(你将得到字节串)或者文本模式(字符串)

一般来说，如果文件格式是由人写的，那么它更可能是文本模式。jpg图像文件一般不是人写的（而且其实不是人直接可读的），因此你应该以二进制模式来打开它们，方法是在mode字符串后加一个b(你可以看看开头的例子里，正确的方式应该是rb)。
如果你以文本模式打开一些东西（比如，加一个t,或者就用r/r+/w/a），你还必须知道要使用哪种编码。对于计算机来说，所有的文件都是字节，而不是字符。

可惜，在Pyhon 2.x版本里，open不支持显示地指定编码。然而，io.open函数在Python 2.x中和3.x(其中它是open的别名)中都有提供，它能做正确的事。你可以传入encoding这个关键字参数来传入编码。
如果你不传入任意编码，一个系统 - 以及Python -指定的默认选项将被选中。你也许被诱惑去依赖这个默认选项，但这个默认选项经常是错误的，或者默认编码实际上不能表达文件里的所有字符（这将经常发生在Python 2.x和/或Windows）。

所以去挑选一个编码吧。utf-8是一个非常好的编码。当你写入一个文件，你可以选一个你喜欢的编码（或者最终读你文件的程序所喜欢的编码）。

那你怎么找出正在读的文件是用哪种编码写的呢？好吧，不幸的是，并没有一个十分简单的方式来检测编码。在不同的编码中，同样的字节可以表示不同，但同样有效的字符。因此，你必须依赖一个元数据（比如，在HTTP头信息里）来找出编码。越来越多的是，文件格式将编码定义成UTF-8。

有了这些基础知识，我们来写一个程序，读取一个文件，检测它是否是JPG（提示：这些文件头部以字节FF D8开始），把对输入文件的描述写入一个文本文件。

```
import io

with open('photo.jpg', 'rb') as inf:
    jpgdata = inf.read()

if jpgdata.startswith(b'\xff\xd8'):
    text = u'This is a JPEG file (%d bytes long)\n'
else:
    text = u'This is a random file (%d bytes long)\n'

with io.open('summary.txt', 'w', encoding='utf-8') as outf:
    outf.write(text % len(jpgdata))
```

我敢肯定，现在你会正确地使用open啦！

目标Python2+3

22. 目标Python2+3

很多时候你可能希望你开发的程序能够同时兼容Python2+和Python3+。

试想你有一个非常出名的Python模块被很多开发者使用着，但并不是所有人都只使用Python2或者Python3。这时候你有两个办法。第一个办法是开发两个模块，针对Python2一个，针对Python3一个。还有一个办法就是调整你现在的代码使其同时兼容Python2和Python3。

本节中，我将介绍一些技巧，让你的脚本同时兼容Python2和Python3。

Future模块导入

第一种也是最重要的方法，就是导入`__future__`模块。它可以帮你在Python2中导入Python3的功能。这有一组例子：

上下文管理器是Python2.6+引入的新特性，如果你想在Python2.5中使用它可以这样做：

```
from __future__ import with_statement
```

在Python3中`print`已经变为一个函数。如果你想在Python2中使用它可以通过`__future__`导入：

```
print
# Output:

from __future__ import print_function
print(print)
# Output: <built-in function print>
```

模块重命名

首先，告诉我你是如何在你的脚本中导入模块的。大多时候我们会这样做：

```
import foo
# or
from foo import bar
```

你知道么，其实你也可以这样做：

```
import foo as foo
```

这样做可以起到和上面代码同样的功能，但最重要的是它能让你的脚本同时兼容Python2和Python3。现在我们来看下面的代码：

```
try:
    import urllib.request as urllib_request  # for Python 3
except ImportError:
    import urllib2 as urllib_request  # for Python 2
```

让我来稍微解释一下上面的代码。

我们将模块导入代码包装在try/except语句中。我们是这样做是因为在Python 2中并没有urllib.request模块。这将引起一个ImportError异常。而在Python2中urllib.request的功能则是由urllib2提供的。所以,当我们试图在Python2中导入urllib.request模块的时候，一旦我们捕获到ImportError我们将通过导入urllib2模块来代替它。

最后，你要了解as关键字的作用。它将导入的模块映射到urllib.request，所以我们通过urllib_request这个别名就可以使用urllib2中的所有类和方法了。

过期的**Python2**内置功能

另一个需要了解的事情就是Python2中有12个内置功能在Python3中已经被移除了。要确保在Python2代码中不要出现这些功能来保证对Python3的兼容。这有一个强制让你放弃12内置功能的方法：

```
from future.builtins.disabled import *
```

现在，只要你尝试在Python3中使用这些被遗弃的模块时，就会抛出一个NameError异常如下：

```
from future.builtins.disabled import *

apply()
# Output: NameError: obsolete Python 2 builtin apply is disabled
```

标准库向下兼容的外部支持

有一些包在非官方的支持下为Python2提供了Python3的功能。例如，我们有：

- enum `pip install enum34`
- singledispatch `pip install singledispatch`
- pathlib `pip install pathlib`

想更多了解，在Python文档中有一个全面的指南可以帮助你让你的代码同时兼容Python2和Python3。

协程

23. 协程

Python中的协程和生成器很相似但又稍有不同。主要区别在于：

- 生成器是数据的生产者
- 协程则是数据的消费者

首先我们先来回顾下生成器的创建过程。我们可以这样去创建一个生成器:

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
```

然后我们经常在for循环中这样使用它:

```
for i in fib():
    print i
```

这样做不仅快而且不会给内存带来压力，因为我们所需要的值都是动态生成的而不是将他们存储在一个列表中。更概括的说如果现在我们在上面的例子中使用yield便可获得了一个协程。协程会消费掉发送给它的值。Python实现的grep就是个很好的例子：

```
def grep(pattern):
    print("Searching for", pattern)
    while True:
        line = (yield)
        if pattern in line:
            print(line)
```

等等！yield返回了什么？啊哈，我们已经把它变成了一个协程。它将不再包含任何初始值，相反要从外部传值给它。我们可以通过send()方法向它传值。这有个例子：

```
search = grep('coroutine')
next(search)
#output: Searching for coroutine
search.send("I love you")
search.send("Don't you love me?")
search.send("I love coroutine instead!")
#output: I love coroutine instead!
```

发送的值会被yield接收。我们为什么要运行next()方法呢？这样做正是为了启动一个协程。就像协程中包含的生成器并不是立刻执行，而是通过next()方法来响应send()方法。因此，你必须通过next()方法来执行yield表达式。

我们可以通过调用close()方法来关闭一个协程。像这样：

```
search = grep('coroutine')
```

```
    search.close()
```

更多协程相关知识的学习大家可以参考David Beazley的这份精彩演讲。

函数缓存

函数缓存 (Function caching)

函数缓存允许我们将一个函数对于给定参数的返回值缓存起来。
当一个I/O密集的函数被频繁使用相同的参数调用的时候，函数缓存可以节约时间。
在Python 3.2版本以前我们只有写一个自定义的实现。在Python 3.2以后版本，有个`lru_cache`的装饰器，允许我们将一个函数的返回值快速地缓存或取消缓存。

我们来看看，Python 3.2前后的版本分别如何使用它。

Python 3.2+

Python 3.2及以后版本

我们来实现一个斐波那契计算器，并使用lru_cache。

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> print([fib(n) for n in range(10)])
# Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

那个maxsize参数是告诉lru_cache，最多缓存最近多少个返回值。

我们也可以轻松地对返回值清空缓存，通过这样：

```
fib.cache_clear()
```

Python 2+

Python 2系列版本

你可以创建任意种类的缓存机制，有若干种方式来达到相同的效果，这完全取决于你的需要。
这里是一个一般的缓存：

```
from functools import wraps

def memoize(function):
    memo = {}
    @wraps(function)
    def wrapper(*args):
        if args in memo:
            return memo[args]
        else:
            rv = function(*args)
            memo[args] = rv
            return rv
    return wrapper

@memoize
def fibonacci(n):
    if n < 2: return n
    return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci(25)
```

这里有一篇Caktus Group的不错的文章，在其中他们发现一个Django框架的由lru_cache导致的bug。读起来很有意思。一定要打开去看一下。

上下文管理器

上下文管理器(Context managers)

上下文管理器允许你在有需要的时候，精确地分配和释放资源。

使用上下文管理器最广泛的案例就是`with`语句了。
想象下你有两个需要结对执行的相关操作，然后还要在它们中间放置一段代码。
上下文管理器就是专门让你做这种事情的。举个例子：

```python
with open('some_file', 'w') as opened_file:
    opened_file.write('Hola!')
```

上面这段代码打开了一个文件，往里面写入了一些数据，然后关闭该文件。如果在往文件写数据时发生异常，它也会尝试去关闭文件。上面那段代码与这一段是等价的：

```python
file = open('some_file', 'w')
try:
    file.write('Hola!')
finally:
    file.close()
```

当与第一个例子对比时，我们可以看到，通过使用`with`，许多样板代码(boilerplate code)被消掉了。 这就是`with`语句的主要优势，它确保我们的文件会被关闭，而不用关注嵌套代码如何退出。

上下文管理器的一个常见用例，是资源的加锁和解锁，以及关闭已打开的文件（就像我已经展示给你看的）。

让我们看看如何来实现我们自己的上下文管理器。这会让我们更完全地理解在这些场景背后都发生着什么。

基于类的实现

基于类的实现

一个上下文管理器的类，最起码要定义__enter__和__exit__方法。
让我们来构造我们自己的开启文件的上下文管理器，并学习下基础知识。

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        self.file_obj.close()
```

通过定义__enter__和__exit__方法，我们可以在with语句里使用它。我们来试试：

```
with File('demo.txt', 'w') as opened_file:
    opened_file.write('Hola!')
```

我们的__exit__函数接受三个参数。这些参数对于每个上下文管理器类中的__exit__方法都是必须的。我们来谈谈在底层都发生了什么。

1. with语句先暂存了File类的__exit__方法
2. 然后它调用File类的__enter__方法
3. __enter__方法打开文件并返回给with语句
4. 打开的文件句柄被传递给opened_file参数
5. 我们使用.write()来写文件
6. with语句调用之前暂存的__exit__方法
7. __exit__方法关闭了文件

处理异常

处理异常

我们还没有谈到__exit__方法的这三个参数：type, value和traceback。
在第4步和第6步之间，如果发生异常，Python会将异常的type,value和traceback传递给__exit__方法。
它让__exit__方法来决定如何关闭文件以及是否需要其他步骤。在我们的案例中，我们并没有注意它们。

那如果我们的文件对象抛出一个异常呢？万一我们尝试访问文件对象的一个不支持的方法。举个例子：

```
with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function('Hola!')
```

我们来列一下，当异常发生时，with语句会采取哪些步骤。

1. 它把异常的type,value和traceback传递给__exit__方法
2. 它让__exit__方法来处理异常
3. 如果__exit__返回的是True，那么这个异常就被优雅地处理了。
4. 如果__exit__返回的是True以外的任何东西，那么这个异常将被with语句抛出。

在我们的案例中，__exit__方法返回的是None(如果没有return语句那么方法会返回None)。因此，with语句抛出了那个异常。

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AttributeError: 'file' object has no attribute 'undefined_function'
```

我们尝试下在__exit__方法中处理异常：

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        print("Exception has been handled")
        self.file_obj.close()
        return True

with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function()

# Output: Exception has been handled
```

我们的__exit__方法返回了True,因此没有异常会被with语句抛出。

这还不是实现上下文管理器的唯一方式。还有一种方式，我们会在下一节中一起看看。

基于生成器的实现

基于生成器的实现

我们还可以用装饰器(decorators)和生成器(generators)来实现上下文管理器。
Python有个`contextlib`模块专门用于这个目的。我们可以使用一个生成器函数来实现一个上下文管理器，而不是使用一个类。
让我们看看一个基本的，没用的例子：

```
from contextlib import contextmanager

@contextmanager
def open_file(name):
    f = open(name, 'w')
    yield f
    f.close()
```

OK啦！这个实现方式看起来更加直观和简单。然而，这个方法需要关于生成器、`yield`和装饰器的一些知识。在这个例子中我们还没有捕捉可能产生的任何异常。它的工作方式和之前的方法大致相同。

让我们小小地剖析下这个方法。

1. Python解释器遇到了`yield`关键字。因为这个缘故它创建了一个生成器而不是一个普通的函数。
2. 因为这个装饰器，`contextmanager`会被调用并传入函数名（`open_file`）作为参数。
3. `contextmanager`函数返回一个以`GeneratorContextManager`对象封装过的生成器。
4. 这个`GeneratorContextManager`被赋值给`open_file`函数，我们实际上是在调用`GeneratorContextManager`对象。

那现在我们既然知道了所有这些，我们可以用这个新生成的上下文管理器了，像这样：

```
with open_file('some_file') as f:
    f.write('hola!')
```

推荐阅读

推荐阅读

本书推送贴的留言和讨论

- v2ex: http://www.v2ex.com/t/267557
- 微博长文: http://weibo.com/1054764633/DoN6Z5Haq?type=repost

v2ex网友florije推荐

- 另外一本同名IntermediatePython的更新的书 https://leanpub.com/intermediatepython

v2ex网友xiaket推荐

- 对于Python提高类的书，推荐Fluent Python 或 Pro Python

v2ex网友shishen10 推荐

- 老齐的教程 https://github.com/qiwsir/StarterLearningPython
- 老齐还整理了很多精华 https://github.com/qiwsir/ITArticles

v2ex网友xiaowangge推荐

Yixiaohan整理了一个不错的推荐：Python初学者（零基础学习Python、Python入门）书籍、视频、资料、社区推荐大家可以前去Fork。

v2ex推荐学习书目

- Learn Python the Hard Way
- Python 学习手册-第五版中文版
- Python Cookbook
- Python 基础教程

捐赠名单

捐赠名单

感谢以下童鞋的捐助，你们的慷慨是我们继续的动力：

特别感谢 言* 的两次慷慨捐赠。

donor	value
好心人	8.80
不知名	6.00
李*雷	8.80
su*z	1.00
Zer*	10.00
牛*甫	5.00
钟*才	16.00
J*m	6.00
鑫*	10.00
黄*	4.99
tho*s	5.00
钱*	2.00
py*	10.00
吼*	5.29
破*	5.00
吉祥*	6.66
Lynd*	8.88
言*	18.00
言*	88.00

你还记得，深夜埋头学习的你吗？